POWER PROGRAMMING

The Windows 3.1 ToolHelp DLL, Part 1

BY RAY DUNCAN

At the inception of its "Blue Ninja" marketing campaign, IBM touted OS/2, Version 2.0, as "a better DOS than DOS, a better Windows than Windows." More recently, IBM has backed down a bit from these claims, but the company is still emphasizing the superior reliability of OS/2 running Windows applications as opposed to Windows running Windows applications. Nobody finds this particularly hard to believe. Just about anyone who runs Windows has experienced at least one or two major crashes, and those of us who develop Windows programs are privileged to watch the entire system regularly fall on its face. I just love it when I type Exit in a Windows DOS box, only to have the screen flash horribly and then find myself again at the DOS prompt—except that Windows and all the applications I was running while I was temporarily using the DOS box are history. This brings up the obvious question: If both OS/2 and Windows run in protected mode, just why is Windows such a fragile environment?

The primary reason, of course, is that Windows is a multitasking, graphical environment layered on top of a single-tasking, character-oriented operating system, and the relationship between the two is not, to put it kindly, elegant. Windows relies on all sorts of undocumented DOS structures and functions, captures many DOS and ROM BIOS interrupt vectors, and inserts legions of little tendrils and hooks into DOS's innards. Moreover, Windows contains its own memory manager and its own suite of device drivers, which must somehow coexist with the DOS memory manager and device drivers. This is a classic example of a statue of gold (well, okay, maybe silver) with feet of clay.

For the first two versions of Windows, there was quicksand under the feet of clay, as well: the Intel 80*x*86 real mode, which required Windows to use a truly Byzantine scheme for virtual memory management and allowed an errant application to scribble anywhere in memory. In spite of this hardware deficiency, the earliest versions of Windows did not even make a token attempt to check the parameters that were passed to an API function by an application before using them! If the application supplied a bad pointer or handle, the function call would not only fail but it could take the whole system down with it. Sometimes, unfortunately, the damage would be subtle enough that the application or the system would continue to run for quite a while before staggering to a halt, making the original cause of the problem nearly impossible to track down.

A sample program lets you explore ToolHelp's facilities for installing a notification callback routine for Windows API parameter errors.

The basic instability of Windows has always been complicated by the fact that Windows applications have tended to be, frankly, very buggy. Try running any large first- or second-generation Windows application on a Windows system with the debugging kernel installed and you'll be astonished at the number of diagnostic messages. Microsoft deserves most of the blame for this. In the earliest days of Windows, the message traffic, the relationships between events, and the interdependencies of Windows API functions were very poorly documented; the debugging facilities were crude; and the debugging hooks were a closely guarded secret, which prevented third parties from supplying more capable debuggers. Consequently, most developers had only a dim understanding of how the system as a whole really fit together, and they built their programs by cutting and pasting source code from other, previously working programs or by cloning source code published by Windows gurus like Charles Petzold and Michael Geary.

Windows 3.0, which was the first version of the operating environment to run in protected mode, brought with it the dreaded UAE (Unrecoverable Application Error). It's important to note, however, that UAEs were (for the most part) not a symptom of new problems with applications and Windows itself; rather they were an unmasking of old problems. In protected mode, use of a bad pointer caused an immediate General Protection fault (a special hardware interrupt) instead of an unpredictable amount of damage and a crash somewhere down the road. The Windows 3.0 developers were apparently so rattled by the whole concept of GP faults that they decided to handle them all in a very conservative way: terminating the active application and displaying a scary dialog box that warned the user to restart his machine immediately or suffer the consequences, even though in the vast majority of cases the system itself was undamaged. Inexpli-

cably, they still made no attempt to perform sanity checks on the parameters of Windows function calls. Windows was probably the only general-purpose multitasking environment in computer history to place such total faith in the correctness of the applications that ran under its control.

Microsoft didn't really get interested in improving the robustness of Windows until after the Great Divorce from IBM, when it suddenly became apparent that the stability of Windows was going to be a crucial competitive issue. During the period between Windows 3.0 and 3.1, Microsoft took many laudable—if sadly overdue—steps to improve life for Windows programmers. A utility called Dr. Watson, which trapped application errors and wrote diagnostic information to a file on-disk, was distributed to developers and users via bulletin boards and online services. A TEST program was provided that allowed programmers to exercise their applications under a variety of stressful conditions. A single-screen debugger was released, and many previously mysterious aspects of the system were cleaned up and documented. Microsoft also implemented a new dynamic link library, called TOOLHELP.DLL, that gave applications access to certain internal Windows structures and made it possible to write debuggers using officially supported interfaces. The ToolHelp functions are summarized in Figure 1. This library was licensed, along with the Windows header files and import libraries, to third-party developers, which set off an explosive evolution in Windows programming tools.

Finally, with the release of Windows 3.1; Microsoft made TOOLHELP.DLL and Dr. Watson an integral part of the retail Windows package and incorporated parameter validation and local reboot into the system. *Parameter validation* means that Windows checks most arguments for an application function call for "reasonable" values before executing the function. If a parameter is bogus, Windows will take what it feels to be an appropriate action. In some cases, it simply ignores the function call or returns an error to the calling program; in others, it gives the user the opportunity to abort the application. *Local reboot* means that Windows traps the Ctrl-Alt-Del key combination and gives the user the option of simply killing the current program rather than resetting the entire system.

THE BADAPP PROGRAM To give you a test-bed for playing around with Windows 3.1's parameter validation, fault trapping, Dr. Watson, and local reboot, I've written a toy program called BADAPP. The C source code for BADAPP is found in Figure 2 (for reasons of space, the header file, module definition file, and resource script are not shown here). BADAPP uses the same table-driven structure for message handling and menu decoding that I introduced in my previous series of columns on the Windows 3.1 Common Dialogs. All of the source files and the executable file for BADAPP can be downloaded as BADAPP.ZIP from the Programming Forum on PC MagNet, or can be obtained by writing *PC Magazine* (see instructions at the end of this column).

The source code for BADAPP is so simple that it should be self-explanatory, a BADAPP is a pretty trivial program from the user's point of view as well. The File pop-up menu has only one item, Exit, that is used to shut down the program in the "normal" manner. The Crash pop-up menu offers several choices: GP Fault, Divide by Zero, Pass Bad Pointer, Infinite Loop, and Pass Bad HDC (device context handle). Each of the Crash menu items tests a different aspect of Windows 3.1's ability to detect and protect itself from severe application bugs. You'll notice some interesting differences in the way Windows handles these five cases.

Parameter validation means that Windows checks most arguments for an application function call for "reasonable" values before executing the function.

Selection of the GP Fault menu item causes control to pass to the DoMenuGPFault() routine, which allocates a far pointer on the stack, initializes it to zero, and then dereferences it. Windows intercepts the resulting GP fault and puts up a Close or Ignore dialog box. If you pick Close, Windows terminates BADAPP unilaterally. If you choose Ignore, Windows allows the application to continue its execution. This has no particularly bad implications in the BADAPP program, because BADAPP doesn't do anything with the data it is fetching via the invalid pointer. A real application would, of course, end up processing garbage if the user picked Ignore, most likely leading to another GP fault or some other catastrophe within a very short time.

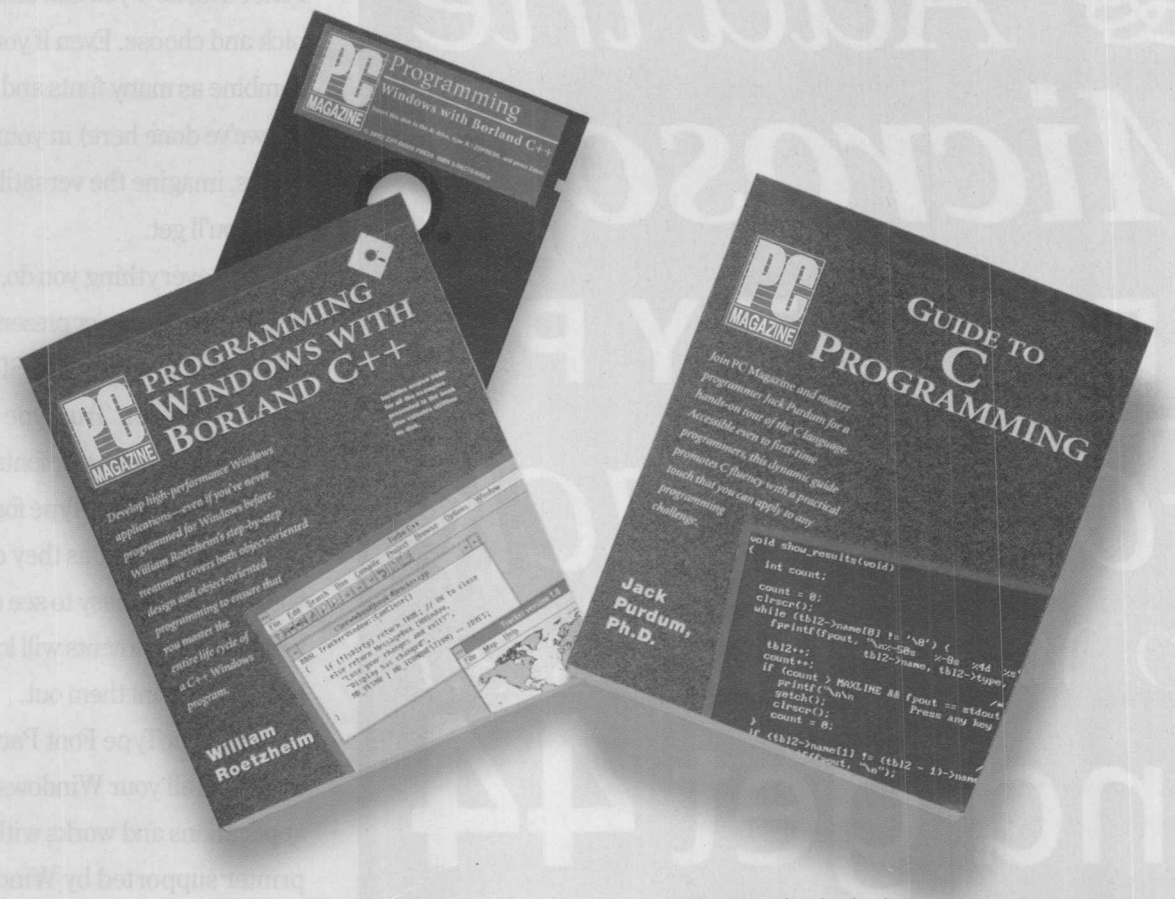
The Divide by Zero and Pass Bad Pointer menu commands cause BADAPP to call the DoMenuZeroDiv() and DoMenuBadPtr() routines, respectively. The first of these routines demands an integer-divide-by-zero operation that Borland C++, for performance reasons, compiles as in-line code without any error

The ToolHelp Functions

Function category	Function names
Window information	ClassFirst(), ClassNext(), GetClassInfo()
Task and module information	ModuleFirst(), ModuleNext(), ModuleFindHandle(), ModuleFindName(), TaskFirst(), TaskNext(), TaskFindHandle()
Memory information	SystemHeapInfo(), MemManInfo()
Global heap information	GlobalInfo(), GlobalFirst(), GlobalNext(), GlobalEntryHandle(), GlobalEntryModule()
Local heap information	LocalInfo(), LocalFirst(), LocalNext()
Stack information()	StackTraceFirst(), StackTraceNext(), StackTraceCSIPFirst()
Execution timing information	TimerCount()
Huge object access	MemoryRead(), MemoryWrite(), GlobalHandleToSel()
Error callbacks	InterruptRegister(), InterruptUnRegister(), NotifyRegister(), NotifyUnRegister()
Process control	TaskGetCSIP(), TaskSetCSIP(), TaskSwitch(), TerminateApp()

Figure 1: **A summary of the functions exported by TOOLHELP.DLL.**

C the Future.



ISBN 1-56276-040-8
Price $39.95

ISBN 1-56276-069-6
Price $29.95

C and C++ have become the languages of choice among today's programmers and future programmers. Whether you're just entering the world of C or moving to the power of C++, the authorities at *PC Magazine* have the knowledge to get you where you want to go.

If you're new to C programming or looking to sharpen your skills, **PC Magazine Guide to C Programming** will get you the results that you want. Internationally acclaimed author, and C instructor Jack Purdum provides a solid foundation in all aspects of C programming. Within no time, Dr. Purdum's unique instructional methods will have you writing functional C code that you can apply on any platform using any C compiler.

Windows programming just got easier. World-renowned expert William Roetzheim shows you how to make Windows Graphical User Interface application development a productive and rewarding experience in **PC Magazine Programming Windows with Borland C++**. This unique book/disk package covers object-oriented programming, the Windows Application Program Interface, and the Object Windows Library, to help C programmers into the power and elegance of Borland C++.

Waldensoftware
Waldenbooks

Visit your local Waldensoftware or Waldenbooks store, or call to order 1-800-322-2000. *PC Magazine Guide to C Programming*: Dept. 597, Item #6781. *PC Magazine Programming Windows with Borland C++*: Dept. 597, Item #6862. Check your yellow pages for the store nearest you.

ZIFF-DAVIS ZD PRESS

CIRCLE 004 ON READER SERVICE CARD

Add the Microsoft TRUETYPE Font Pack TO WINDOWS 3.1 and get 44 New Ways TO LOOK GOOD on paper.

With the Microsoft TrueType® Font Pack, now you can afford to pick and choose. Even if you don't combine as many fonts and icons (as we've done here) in your documents, imagine the versatility and style you'll get.

Now everything you do, from quick memos to major presentations, will benefit from true WYSIWYG precision output. TrueType lets you dynamically scale your fonts to any size. And since TrueType fonts look the same on paper as they do on your screen, it's easy to see exactly what your documents will look like before you print them out.

The TrueType Font Pack upgrades all your Windows-based applications and works with every printer supported by Windows™ operating system version 3.1.

What's more, the TrueType Font Pack completes the set of standard fonts included with most PostScript® printers. Just install and go.

Isn't it time you added the Microsoft TrueType Font Pack for Windows? For more information or the name of a reseller near you, call (800) 541-1261, Ext. HC6.

Microsoft
Making it easier

MICROSOFT WINDOWS

hecking. The second routine calls the Windows API function MessageBox with a null pointer for the string to be displayed within the message box. In both cases, Windows shuts down the application immediately without any option to continue. The handling of Pass Bad Pointer is a little surprising, because Windows could use the Intel 80286 VERR instruction to check out the pointer "for free" before trying to use it, and just fail the function call if the pointer was invalid. Apparently, however, Windows just dereferences the pointer blindly, a GP fault occurs within the Windows kernel, and all this leads to a UAE condition for the application.

If you pick Infinite Loop, BADAPP calls its DoMenuHang routine, which consists mainly of a while(TRUE){} structure, and the entire system enters a funny state. The screen looks okay and the mouse pointer will still move around on the screen, but nothing else works. This is because the mouse generates interrupts, and interrupts are not masked (since the infinite loop occurs at the application level rather than within the Windows kernel), so the mouse driver can continue to monitor the mouse's location and update the screen accordingly. All other Windows activities, however, necessarily come to a screeching halt because Windows multitasking is cooperative rather than preemptive. If the application is in an infinite loop, it never processes its own queue messages or yields control, and other applications never get a chance to process messages, either. If you press the Ctrl-Alt-Del key combination, a keyboard interrupt occurs, the keyboard driver gets control, and it calls the Windows local reboot routine, which gives you the choice to continue execution, abort the current appli-

BADAPP.C

1 of 2

```
// BADAPP - Application Error Testbed for Windows 3.1
// Copyright (C) 1992 Ray Duncan
// Ziff Davis Publishing * PC Magazine

#define dim(x) (sizeof(x) / sizeof(x[0]))   // returns no. of elements

#include "windows.h"
#include "badapp.h"

HANDLE hInst;                               // module instance handle
HWND hFrame;                                // handle for frame window

char szShortAppName[] = "BadApp";           // short application name
char szAppName[] = "BadApp - UAE Tester";   // long application name
char szMenuName[] = "BadAppMenu";           // name of menu resource

struct decodeWord {                         // structure associates
    UINT Code;                              // messages or menu IDs
    LONG (*Fxn)(HWND, UINT, UINT, LONG); }; // with a function

//
// Table of window messages supported by FrameWndProc()
// and the functions which correspond to each message.
//
struct decodeWord messages[] = {
    WM_PAINT, DoPaint,
    WM_COMMAND, DoCommand,
    WM_DESTROY, DoDestroy, } ;

//
// Table of menubar item IDs and their corresponding functions.
//
struct decodeWord menuitems[] = {
    IDM_EXIT, DoMenuExit,
    IDM_GPFAULT, DoMenuGPFault,
    IDM_HANG, DoMenuHang,
    IDM_ZERODIV, DoMenuZeroDiv,
    IDM_BADHDC, DoMenuBadHDC,
    IDM_BADPTR, DoMenuBadPtr, } ;

//
// WinMain -- entry point for this application from Windows.
//
INT APIENTRY WinMain(HANDLE hInstance,
    HANDLE hPrevInstance, LPSTR lpCmdLine, INT nCmdShow)
{
    MSG msg;

    hInst = hInstance;                      // save this instance handle

    if(!hPrevInstance)                      // if first instance,
        if(!InitApplication(hInstance))     // register window class
            return(FALSE);                  // exit if couldn't register

    if(!InitInstance(hInstance, nCmdShow))  // create this instance's window
        return(FALSE);                      // exit if create failed

    while(GetMessage(&msg, NULL, 0, 0))     // while message != WM_QUIT
    {
        TranslateMessage(&msg);             // translate virtual key codes
        DispatchMessage(&msg);              // dispatch message to window
    }

    TermInstance(hInstance);                // clean up for this instance
    return(msg.wParam);                     // return code = WM_QUIT value
}

//
// InitApplication --- global initialization code for this application.
//
BOOL InitApplication(HANDLE hInstance)
{
    WNDCLASS  wc;

    // set parameters for frame window class
    wc.style = CS_HREDRAW | CS_VREDRAW;         // class style
    wc.lpfnWndProc = FrameWndProc;              // class callback function
    wc.cbClsExtra = 0;                          // extra per-class data
    wc.cbWndExtra = 0;                          // extra per-window data
    wc.hInstance = hInstance;                   // handle of class owner
    wc.hIcon = LoadIcon(hInst, "BadAppIcon");   // skull icon
    wc.hCursor = LoadCursor(NULL, IDC_ARROW);   // default cursor
    wc.hbrBackground = GetStockObject(WHITE_BRUSH); // background color
    wc.lpszMenuName =  szMenuName;              // name of menu resource
    wc.lpszClassName = szShortAppName;          // name of window class

    return(RegisterClass(&wc));                 // register class, return status
}

//
// InitInstance --- instance initialization code for this application.
//
BOOL InitInstance(HANDLE hInstance, int nCmdShow)
{
    hFrame = CreateWindow(                      // create frame window
        szShortAppName,                         // window class name
        szAppName,                              // text for title bar
        WS_OVERLAPPEDWINDOW,                    // window style
        CW_USEDEFAULT, CW_USEDEFAULT,           // default position
        CW_USEDEFAULT, CW_USEDEFAULT,           // default size
        NULL,                                   // no parent window
        NULL,                                   // use class default menu
        hInstance,                              // window owner
        NULL);                                  // unused pointer

    if(!hFrame) return(FALSE);                  // error, can't create window

    ShowWindow(hFrame, nCmdShow);               // make frame window visible
    UpdateWindow(hFrame);                       // force WM_PAINT message
    return(TRUE);                               // return success flag
}

//
// TermInstance -- instance termination code for this application.
//
BOOL TermInstance(HANDLE hinstance)
{
    return(TRUE);                               // return success flag
}

//
// FrameWndProc --- callback function for application frame window.
//
LONG FAR APIENTRY FrameWndProc(HWND hWnd, UINT wMsg, UINT wParam, LONG lParam)
{
    int i;                                      // scratch variable

    for(i = 0; i < dim(messages); i++)          // decode window message and
    {                                           // run corresponding function
        if(wMsg == messages[i].Code)
            return((*messages[i].Fxn)(hWnd, wMsg, wParam, lParam));
    }

    return(DefWindowProc(hWnd, wMsg, wParam, lParam));
}

//
// DoCommand -- process WM_COMMAND message for frame window by
// decoding the menubar item with the menuitems[] array, then
// running the corresponding function to process the command.
```

Figure 2: **BADAPP.C, the C-language source code for the BADAPP demonstration program.**

BADAPP.C

2 of 2

```
//
LONG DoCommand(HWND hWnd, UINT wMsg, UINT wParam, LONG lParam)
{
    int i;                                  // scratch variable

    for(i = 0; i < dim(menuitems); i++)     // decode menu command and
    {                                       // run corresponding function
        if(wParam == menuitems[i].Code)
            return((*menuitems[i].Fxn)(hWnd, wMsg, wParam, lParam));
    }

    return(DefWindowProc(hWnd, wMsg, wParam, lParam));
}

//
// DoDestroy -- process WM_DESTROY message for frame window.
//
LONG DoDestroy(HWND hWnd, UINT wMsg, UINT wParam, LONG lParam)
{
    PostQuitMessage(0);                     // force WM_QUIT message to
    return(FALSE);                          // terminate the event loop
}

//
// DoPaint -- process WM_PAINT message for frame window.  Select
// a pretty font, then display a message in the center of the window
// to show that the program is alive.
//
LONG DoPaint(HWND hWnd, UINT wMsg, UINT wParam, LONG lParam)
{
    HDC hdc;
    PAINTSTRUCT ps;
    RECT rect;
    HFONT hfont;

    hdc = BeginPaint(hWnd, &ps);            // get device context
    GetClientRect(hWnd, &rect);             // get client area dimensions
    hfont = CreateFont(-36, 0, 0, 0, 700, TRUE, 0, 0, ANSI_CHARSET,
        OUT_DEFAULT_PRECIS, CLIP_DEFAULT_PRECIS,
        DEFAULT_QUALITY, (FF_MODERN << 4) + DEFAULT_PITCH,
        "Ariel");
    SelectObject(hdc, hfont);               // select new font
    DrawText(hdc, "I'm Really Bad!", -1,    // paint text in window
        &rect, DT_CENTER | DT_VCENTER | DT_SINGLELINE);
    EndPaint(hWnd, &ps);                    // release device context
    return(FALSE);
}

//
// DoMenuExit -- process File-Exit command from menu bar.
//
LONG DoMenuExit(HWND hWnd, UINT wMsg, UINT wParam, LONG lParam)
{
    SendMessage (hWnd, WM_CLOSE, 0, 0L);    // send window close message
    return(FALSE);                          // to shut down the app
}

//
// DoMenuGPFault -- process GP Fault command from menu bar.  Force
// a fault by dereferencing a null pointer.
//
LONG DoMenuGPFault(HWND hWnd, UINT wMsg, UINT wParam, LONG lParam)
{
    char c;
    char far *p = NULL;

    c = *p;                                 // dereference bogus pointer
    return(FALSE);                          // to force a GP fault
}

//
// DoMenuHang -- process Infinite Loop command from menu bar.
// Execute an infinite loop until the user forces a local reboot
// with Ctl-Alt-Del.
//
LONG DoMenuHang(HWND hWnd, UINT wMsg, UINT wParam, LONG lParam)
{
    while(TRUE)
    {
        // execute practically forever
    }

    return(FALSE);
}

//
// DoMenuZeroDiv -- process Divide by Zero command from menu bar.
// Declare some integers, then force a divide by zero fault.
//
LONG DoMenuZeroDiv(HWND hWnd, UINT wMsg, UINT wParam, LONG lParam)
{
    int i, j, k = 0;

    i = j / k;
    return(FALSE);
}

//
// DoMenuBadHDC -- process Bad HDC command from menu bar.  Call the
// DrawText() API function with a bogus HDC.
//
LONG DoMenuBadHDC(HWND hWnd, UINT wMsg, UINT wParam, LONG lParam)
{
    HDC hdc = 0;
    RECT rect;

    GetClientRect(hWnd, &rect);             // get client area dimensions
    DrawText(hdc, "Bogus HDC!", -1,         // use invalid HDC to paint text
        &rect, DT_CENTER | DT_VCENTER | DT_SINGLELINE);
    return(FALSE);
}

//
// DoMenuBadPtr -- process Bad Pointer command from menu bar.  Call the
// MessageBox() API function with a NULL pointer to the message string.
//
LONG DoMenuBadPtr(HWND hWnd, UINT wMsg, UINT wParam, LONG lParam)
{
    MessageBox(hWnd, NULL, NULL, MB_OK | MB_ICONSTOP);
    return(FALSE);
}
```

cation, or reset the machine entirely.

How about the Pass Bad HDC menu item? The corresponding BADAPP routine DoMenuBadHDC is implemented as a call to DrawText with a device context handle (HDC) of zero, and it appears to do nothing. The Windows developers apparently decided that this particular error fell into the category where they could safely ignore the function call and allow the application to continue. After all, the worst that could happen is that the user wouldn't see something on the screen that ought to be there. If you are running the special debugging Windows kernel, however, you will get a warning message about the invalid handle on the debugging terminal.

WORKING WITH TOOLHELP ToolHelp offers quite a diverse range of services, many of which we will be using in demonstration programs in this column over the next few issues. Just to get our feet wet, however, let's try out ToolHelp's facilities for installing a *notification callback routine* for Windows API parameter errors. Whenever the application requests a Windows function with an invalid parameter, Windows will (while the function call is still in progress) call back into the notification routine with information about the error. This allows the application to decide what to do about its own error, but more important, allows the programmer to run the application under completely natural conditions—no debugger, no separate debugging terminal, no special Windows debugging kernel—and still gather information about this particular class of bugs.

In order to use a notification callback routine in your own program, you must do the following:

- Register an application-specific window message in your application's instance initialization routine; this window message will be used by the notification callback routine to communicate with the body of your program.
- Allocate a thunk for the callback routine with MakeProcInstance() in the program's instance initialization routine.
- Register the notification callback routine with TOOLHELP.DLL by calling the function NotifyRegister() in the program's instance initialization routine.
- Implement the notification callback routine itself. This entry must be declared as BOOL FAR PASCAL. It accepts an unsigned single integer and an unsigned

BADAPP2.C

Complete Listing

```
// BADAPP2 - Application Error Testbed for Windows 3.1
//           with Notification Callback
// Copyright (C) 1992 Ray Duncan
// Ziff Davis Publishing * PC Magazine

#include "windows.h"
#include "toolhelp.h"
#include "badapp.h"
  .
  .
  .
WNDPROC lpNotifyCallback;               // NotifyCallback thunk addr
  .
  .
  .
//
// Table of window messages supported by FrameWndProc()
// and the functions which correspond to each message.
//
struct decodeWord messages[] = {
    0, DoNotifyMessage,
    WM_PAINT, DoPaint,
    WM_COMMAND, DoCommand,
    WM_DESTROY, DoDestroy, } ;
  .
  .
  .
//
// These structures receive error information during ToolHelp callback.
//
NFYLOGERROR nfylogerror;
NFYLOGPARAMERROR nfylogparamerror;
  .
  .
  .
//
// InitInstance --- instance initialization code for this application.
//
BOOL InitInstance(HANDLE hInstance, int nCmdShow)
{
  .
  .
  .
    // allocate private message number for use by NotifyCallback
    messages[0].Code = RegisterWindowMessage("BADAPP");

    // allocate thunk for callback then register it with ToolHelp.DLL
    lpNotifyCallback = MakeProcInstance((WNDPROC)NotifyCallback, hInst);
    NotifyRegister(NULL, lpNotifyCallback, NF_NORMAL);

    return(TRUE);                       // return success flag
}

//
// TermInstance -- instance termination code for this application.
//
BOOL TermInstance(HANDLE hinstance)
{
    NotifyUnRegister(NULL);             // unregister ToolHelp callback
    FreeProcInstance(lpNotifyCallback); // release callback
    return(TRUE);                       // return success flag
}
  .
  .
//
// DoNotifyMessage -- process private message from NotifyCallback
//
LONG DoNotifyMessage(HWND hWnd, UINT wMsg, UINT wParam, LONG lParam)
{
    UINT i;
    char *p;
    char temp[256];

    switch(wParam)
    {
        case NFY_LOGERROR:
            p = "NFY_LOGERROR";
            i = nfylogerror.wErrCode;
            break;

        case NFY_LOGPARAMERROR:
            p = "NFY_LOGPARAMERROR";
            i = nfylogparamerror.wErrCode;
            break;

        default:
            p = "Unknown";
            i = 0;
    }

    // now format and display NotifyCallback information
    wsprintf(temp, "Notify type: %s\nError code: %04Xh", (LPSTR) p, i);
    MessageBox(hWnd, temp, "BadApp", MB_OK | MB_ICONEXCLAMATION);

    return(FALSE);
}
  .
  .
  .
//
// NotifyCallback -- callback routine for ToolHelp DLL.  Looks for
// codes indicating a parameter validation error in an API call,
// saves information about the error, and posts a message to the
// main message handling routine indicating that an error occurred.
//
BOOL FAR APIENTRY NotifyCallback(UINT wID, LONG dwData)
{
    switch(wID)
    {
        case NFY_LOGERROR:              // API parameter error
            nfylogerror = *(NFYLOGERROR far *) dwData;
            PostMessage(hFrame, messages[0].Code, wID, 0);
            return(TRUE);               // signal callback was handled

        case NFY_LOGPARAMERROR:         // API parameter error
            nfylogparamerror = *(NFYLOGPARAMERROR far *) dwData;
            PostMessage(hFrame, messages[0].Code, wID, 0);
            return(TRUE);               // signal callback was handled

        default:
            return(FALSE);              // signal callback not handled
    }
}
```

Figure 3: **Extracts from BADAPP2.C, a modified version of BADAPP with notification callback handling. The full source code is available on PC MagNet.**

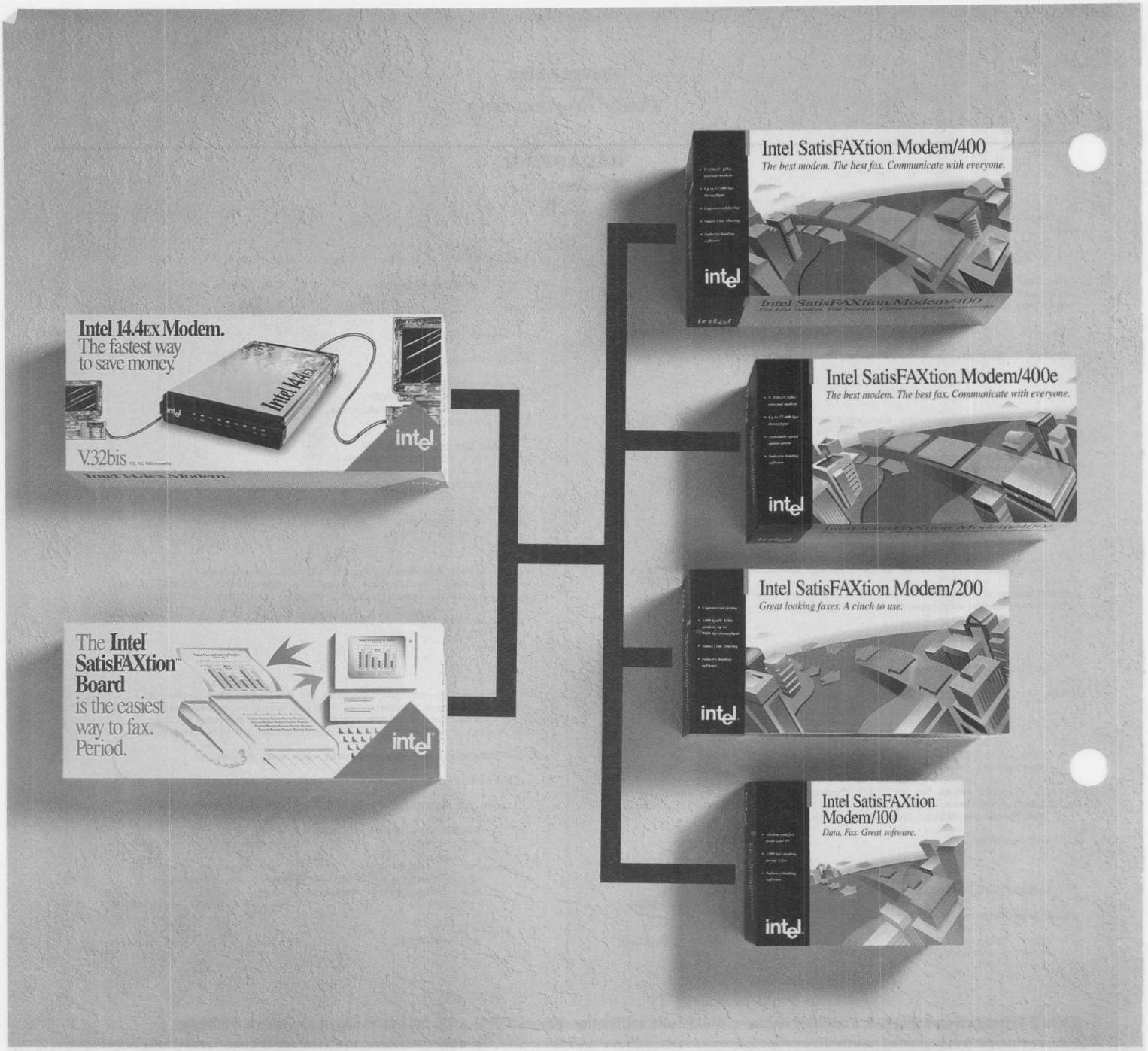
double integer as parameters; the single value is an error code and the double value is usually a far pointer to a structure that contains more information about the error. The callback routine is not allowed to do very much except call ToolHelp functions, make a copy of the error information in local static variables, and post a message to one of the application's window handlers.

- Add appropriate logic to one of the application's window handlers to process the message posted by the notification callback and take appropriate action (for example, log the error to a disk file or display an alert dialog).
- Modify the application's termination code to call the ToolHelp function NotifyUnRegister; this allows TOOLHELP.DLL to deallocate those of its internal data structures that are associated with the application.
- Add static NFYLOGERROR or NFYLOGPARAMERROR structures to the program; these can be used by the notification callback routine to make local copies of the error information that is passed to it.
- Export the notification callback routine in the application's module definition (.DEF) file.

To illustrate these steps, I have modified the BADAPP.C source code to create a new program, BADAPP2.C, which contains a notification callback. We could not print the entire listing, but extracts of BADAPP2's source code, showing the portions that are specific to the handling of notification callbacks, can be found in Figure 3. As always, you can get the full source code and executable file for BADAPP2.C by downloading it from PC MagNet, or by sending a postcard with your name and address to the attention of Katherine West, PC Magazine, One Park Ave., New York, NY 10016; (no phone calls, please).

THE IN-BOX Please send your questions, comments, and suggestions to me at any of the following electronic mail addresses:

PC MagNet: 72241,52
MCI Mail: rduncan
BIX: rduncan □



Married. With children.

The traditional modem is a dying breed. And in its place comes a new generation of products.

Introducing the Intel family of faxmodems—a combination of our award-winning modem technology and SatisFAXtion® Board that brings you a no-compromise modem and fax in one. All for about the price of a data modem alone.

Our top-of-the-line product, the SatisFAXtion® Modem/400, is 50 percent faster than a traditional 9600bps modem. Plus it features unequalled fax capabilities. The Model 400e is the powerful external brother to the Model 400. And Models 200 and 100 are excellent values if you have less need for speed.

So don't settle for just a modem. For the ultimate performance, features and compatibility only Intel can give you, tie the faxmodem knot now.

Experience our no-obligation, 14-day Test Drive program. For the Intel Test Drive dealer nearest you, call 1-800-538-3373, ext. 61.

WE MAKE COMPUTERS WORK HARDER.

Find out more. Ask for ext. 61
1-800-538-3373

intel®

POWER PROGRAMMING

The Windows 3.1 ToolHelp DLL, Part 2

BY RAY DUNCAN

In the last column, I introduced you to Windows 3.1's TOOLHELP .DLL, a dynamic link library that gives you access to previously undocumented Windows data structures and function calls. I had intended to continue in this issue with an example of how to trap General Protection (GP) Faults and other hardware exceptions using TOOLHELP.DLL. But, to be frank, I couldn't get the code into a form that I considered publishable in time. ToolHelp's support for interrupt handling is sufficient to deliver control to your program's routines, but it's inadequate for clean handling of the interrupt. Furthermore, the ToolHelp support is not structured so that you can easily write the entire interrupt handler in a high-level language. I can get the stuff to work, but my current solution just isn't aesthetically appealing.

So I'll let my subconscious work on the ToolHelp interrupt-handler problem for awhile, and look at a different topic this time: the ToolHelp API for inspecting and enumerating the system module list, task list, window classes, and global heap. If you're an experienced Windows programmer, this API will be a pleasant surprise: It's predictable, symmetric, and easy to use; the function names actually make some kind of sense; and it's powerful enough for the needs of just about any system-spelunking application.

THE SYSTEM VOYEUR API I think of the ToolHelp functions related to the task list, module list, window class list, and global heap as "voyeur" functions because they let you peek at, but not touch, the corresponding Windows internal data structures. Actually, you don't even get a direct look at the structures—you get a transformed and somewhat massaged reflection of the actual data, copied into data structures that lie in your application's own address space. The functions fall into two categories (as shown in Figure 1): those that let you "walk" through one of the system's linked lists of data objects, and those that let you retrieve information about a specific data object using a handle or name.

All of the ToolHelp voyeur functions

A sample program demonstrates the ToolHelp API for inspecting the system module list, task list, window classes, and global heap.

are used in essentially the same way. First, you allocate a data structure specific to the type of information being retrieved: task, module, window class, or global heap block. Next, you enter the function via a far call, passing a far pointer to the data structure and—if you're retrieving information about a particular object rather than walking a list—an additional flag, handle, or string pointer. All of the functions return a TRUE value if successful and a FALSE value if they fail or encounter an error; other information, if any, is placed in the data structure whose address was passed in the original call.

To make this more concrete, let's see how we would enumerate all the active tasks in the system. We need a data structure called TASKENTRY and two API functions: TaskFirst() and TaskNext(). The data structure must be initialized before use, and the first field of the structure contains the length of the structure, which is used as a sanity check by Windows. The skeleton for a loop that walked the task list would look like this:

```
// allocate the data structure

TASKENTRY te;

//initialize the data structure
memset(&te, 0, sizeof(TASKENTRY));
te.dwSize = sizeof(TASKENTRY);

// now walk the task list
TaskFirst(&te)
do {
   // process results in
   // structure "te" here
 } while(TaskNext(&te));
```

Notice that we don't have to worry about the success of the TaskFirst() call because we know that at least one task will always be running: our own program! The functions for walking the module, window, and global heap lists are used in an almost identical fashion.

Now suppose we want to retrieve the information about a specific task directly, rather than walking the task list until we run into it (or not). Assuming we have the task's handle to work with, we can use the TaskFindHandle() function together with the familiar TASKENTRY data structure:

```
// task info data structure
TASKENTRY te;

// handle of task to find
HTASK htask;
```

```
//initialize the data structure
memset(&te, 0, sizeof(TASKENTRY);
te.dwSize = sizeof(TASKENTRY);

// now retrieve task info
if(TaskFindHandle(&te, htask))
  // function was successful
else
  // an error occurred
```

The API functions for retrieving information about a specific module, class, or global heap block work in much the same way (allowing for the data object–specific structures, of course). The functions for the different data object types can also be used in combination; for example, as you walk down the task list, you can use the module handle returned for a task in the TASKENTRY structure together with the ModuleFindHandle() function to fetch the task's ASCIIZ modulename and pathname. Similarly, when walking the global heap, you can use the owner handle returned in the GLOBALENTRY structure for a particular memory block together with the TaskFindHandle() and ModuleFindHandle() functions to find the task or module name of the block's owner.

THE SYSMON PROGRAM With the advent of TOOLHELP.DLL, the creation of programs that used to require weeks of painful experimentation and debugging is now easy and even fun. For this issue, I've written a simple system monitor utility called SYSMON that lets you display all of the system lists we've discussed: tasks, modules, window classes, and the global heap. The C-language source code for SYSMON is listed in Figure 2, and the complete set of files necessary to build the application, along with the executable file, can be downloaded from PC MagNet, archived as SYSMON.ZIP.

From the user's point of view, SYSMON is just a resizeable window containing scrollable text, with three choices available on the menu bar. The first two menu bar items are pop-ups: the File menu, which lets the user activate the About. . . dialog box or exit the application; and the Display menu, which allows the user to select a display of the task list, module list, window class list, or global heap. (Users with really big screens also have the option of loading four different copies of the program at the same time and having each one display different information.) The third menu item is Refresh!, which simply forces the program to rewalk the currently selected system list and rebuild its output. The observant user will note that SYSMON also automatically refreshes its display every 10 seconds when it doesn't have the input focus, and that it "remembers" its most recent window size and position from one invocation to the next. The screen shot in Figure 3 shows SYSMON in action.

At the source-code level, SYSMON is based on the same table-driven approach to message handling that I showed in the DLGDEMO and BADAPP programs. This scheme facilitates the use of short, reusable routines rather than the lengthy, convoluted case statements generally found in Windows programming books, and I've found that it significantly shortens and simplifies both the coding and debugging phases of writing a new Windows application. When you want to expand a program with code that handles an additional message, you simply write a short routine for that specific message, add the routine's address and the message identifier to the master message handler table, and stick another prototype into the application's header file. The previously existing (and presumably correct) application source code does not need to be changed at all.

As far as the ToolHelp aspects of SYSMON are concerned, the four routines to study are WalkClassList(), WalkTaskList(), WalkModuleList(), and WalkGlobalHeap(). These illustrate calls to the *xxx*First() and *xxx*Next() functions listed in Figure 1, along with their respective data structures. In order to keep the code in these functions easy to understand, I've kept the processing of the data structures rather simple, but you can quickly enhance the functions to format the information in the data structures more completely and more elaborately.

Aside from its interface with TOOLHELP.DLL, SYSMON's source code has several other interesting features you may wish to incorporate into your own programs. The first is SYSMON's capability to recall where it was last located on the screen. This feature turns out to be quite easy to implement, considering the huge difference in friendliness it projects to the user. You just use the functions GetWindowRect() and WritePrivateProfileString() to determine the window coordinates at application termination time and write them into an application-specific .INI file, then use GetPrivateProfileInt() and MoveWindow() to retrieve the coordinates from the .INI file and size and position the window at application initialization time. The SYSMON source code that handles this can be found in the routines UpdateProfile() and InitInstance(), respectively.

The next feature that is especially worth mentioning is the periodic refresh of the display, which is also easy to implement. A thunk for the timer callback function is allocated in InitInstance() with a call to MakeProcInstance(), and then a Windows system timer is activated and its interval specified with a call to SetTimer(). Windows subsequently enters the timer callback function, named TimerProc(), at 10-second intervals—irrespective of what other application might be in the foreground. The callback function first checks whether SYSMON's window is iconized or has the input focus; if neither condition is the case, the callback function just calls SendMessage() to send an IDM_REFRESH message to

The ToolHelp Voyeur Functions

Information type	Walk API	Specific object API	Data structure
Window classes	ClassFirst()	GetClassInfo()	CLASSENTRY
	ClassNext()		
Active modules	ModuleFirst()	ModuleFindHandle()	MODULEENTRY
	ModuleNext()	ModuleFindName()	
Active tasks	TaskFirst()	TaskFindHandle()	TASKENTRY
	TaskNext()		
Global heap	GlobalFirst()	GlobalInfo()	GLOBALENTRY
	GlobalNext()	GlobalEntryHandle()	
	GlobalEntryModule()		

Figure 1: **These functions fall into two categories: those that allow you to "walk" through one of the system's linked lists of data objects, and those that let you retrieve information about a specific data object.**

SYSMON.C

1 of 4

```
// SysMon - System Monitor for Windows 3.1
// Copyright (C) 1992 Ray Duncan
// PC Magazine * Ziff Davis Publishing

#define dim(x) (sizeof(x) / sizeof(x[0]))   // returns no. of elements
#define MAXLINES 4096                       // max lines to display

#include "stdlib.h"
#include "windows.h"
#include "toolhelp.h"
#include "sysmon.h"

HANDLE hInst;                               // module instance handle
HWND hFrame;                                // handle for frame window
HFONT hFont;                                // handle for nonprop. font

int CharX, CharY;                           // character dimensions
int LinesPerPage;                           // lines per page
int CurLine = 0;                            // first line, current page
int TotLines = 0;                           // total lines to display
int TopLine = 0;                            // first line of last page
int DisplayType = IDM_MODULE;               // type of info to display

char *LinePtr[MAXLINES];                    // holds pointers to lines

char szFrameClass[] = "SysMon";             // classname for frame window
char szAppName[] = "System Monitor";        // long application name
char szMenuName[] = "SysMonMenu";           // name of menu resource
char szIni[] = "sysmon.ini";                // name of private INI file

WNDPROC lpTimerProc;                        // timer callback thunk

//
// Table of window messages supported by FrameWndProc()
// and the functions which correspond to each message.
//
struct decodeWord frameMsgs[] = {
    WM_PAINT, DoPaint,
    WM_SIZE, DoSize,
    WM_COMMAND, DoCommand,
    WM_CLOSE, DoClose,
    WM_DESTROY, DoDestroy,
    WM_VSCROLL, DoVScroll, } ;

//
// Table of menubar item IDs and their corresponding functions.
//
struct decodeWord menuitems[] = {
    IDM_EXIT, DoMenuExit,
    IDM_ABOUT, DoMenuAbout,
    IDM_MODULE, DoDisplayType,
    IDM_CLASS, DoDisplayType,
    IDM_TASK, DoDisplayType,
    IDM_HEAP, DoDisplayType,
    IDM_REFRESH, DoRefresh, } ;

//
// WinMain -- entry point for this application from Windows.
//
int APIENTRY WinMain(HANDLE hInstance,
    HANDLE hPrevInstance, LPSTR lpCmdLine, int nCmdShow)
{
    MSG msg;                                // scratch message storage
    hInst = hInstance;                      // save this instance handle

    if(!hPrevInstance)                      // if first instance,
        if(!InitApp(hInstance))             // register window class
        {
            MessageBox(hFrame, "Can't initialize SysMon!", szAppName,
                MB_ICONSTOP | MB_OK);
            return(FALSE);
        }

    if(!InitInstance(hInstance, nCmdShow))  // create this instance's window
    {
        MessageBox(hFrame, "Can't initialize SysMon!", szAppName,
            MB_ICONSTOP | MB_OK);
        return(FALSE);
    }

    while(GetMessage(&msg, NULL, 0, 0))     // while message != WM_QUIT
    {
        TranslateMessage(&msg);             // translate virtual key codes
        DispatchMessage(&msg);              // dispatch message to window
    }

    TermInstance(hInstance);                // clean up for this instance
    return(msg.wParam);                     // return code = WM_QUIT value
}

//
// InitApp --- global initialization code for this application.
//
BOOL InitApp(HANDLE hInstance)
{
    WNDCLASS  wc;                           // window class info

    // set parameters for frame window class
    wc.style = CS_HREDRAW | CS_VREDRAW;     // class style
    wc.lpfnWndProc = FrameWndProc;          // class callback function
    wc.cbClsExtra = 0;                      // extra per-class data
    wc.cbWndExtra = 0;                      // extra per-window data
    wc.hInstance = hInstance;               // handle of class owner
    wc.hIcon = LoadIcon(hInst, "SysMonIcon");      // application icon
    wc.hCursor = LoadCursor(NULL, IDC_ARROW);      // default cursor
    wc.hbrBackground = GetStockObject(WHITE_BRUSH); // background color
    wc.lpszMenuName =  szMenuName;          // name of menu resource
    wc.lpszClassName = szFrameClass;        // name of window class

    return(RegisterClass(&wc));             // register frame window class
}

//
// InitInstance --- instance initialization code for this application.
//
BOOL InitInstance(HANDLE hInstance, int nCmdShow)
{
    HDC hdc;                                // handle for device context
    TEXTMETRIC tm;                          // info about font
    RECT rect;                              // window position & size
    int i;                                  // scratch variable

    for(i = 0; i < MAXLINES; i++)           // initialize all line
        LinePtr[i] = NULL;                  // pointers

    hFrame = CreateWindow(                  // create frame window
        szFrameClass,                       // window class name
        szAppName,                          // text for title bar
        WS_OVERLAPPEDWINDOW | WS_VSCROLL,   // window style
        CW_USEDEFAULT, CW_USEDEFAULT,       // default position
        CW_USEDEFAULT, CW_USEDEFAULT,       // default size
        NULL,                               // no parent window
        NULL,                               // use class default menu
        hInstance,                          // window owner
        NULL);                              // unused pointer

    if(!hFrame) return(FALSE);              // error, can't create window

    hdc = GetDC(hFrame);                    // get device context
    hFont = GetStockObject(SYSTEM_FIXED_FONT);  // handle for nonprop. font
    SelectObject(hdc, hFont);               // realize the font and get
    GetTextMetrics(hdc, &tm);               // the character dimensions
    CharX = tm.tmAveCharWidth;
    CharY = tm.tmHeight + tm.tmExternalLeading;
    ReleaseDC(hFrame, hdc);                 // release device context

    GetWindowRect(hFrame, &rect);           // current window pos & size

    // read profile for frame window from previous invocation, if any
    rect.left   = GetPrivateProfileInt("Frame", "xul", rect.left, szIni);
    rect.top    = GetPrivateProfileInt("Frame", "yul", rect.top, szIni);
    rect.right  = GetPrivateProfileInt("Frame", "xlr", rect.right, szIni);
    rect.bottom = GetPrivateProfileInt("Frame", "ylr", rect.bottom, szIni);

    MoveWindow(hFrame, rect.left, rect.top, // force window size & position
        rect.right-rect.left, rect.bottom-rect.top, TRUE);

    // get display type from previous invocation, default to module list
    DisplayType = GetPrivateProfileInt("Frame", "type", IDM_MODULE, szIni);

    ShowWindow(hFrame, nCmdShow);           // make frame window visible
    UpdateWindow(hFrame);                   // force WM_PAINT message
    SendMessage(hFrame, WM_COMMAND, DisplayType, 0); // initialize display

    // allocate thunk for timer callback routine
    lpTimerProc  = MakeProcInstance((WNDPROC) TimerProc, hInst);

    // set up our 10 sec. (10,000 msec) timer callback
    if (!SetTimer(hFrame, 1, 10000, lpTimerProc))
        return(FALSE);

    return(TRUE);                           // return success flag
}

//
// TermInstance -- instance termination code for this application.
// Does nothing in this case, included for symmetry with InitInstance.
//
BOOL TermInstance(HANDLE hinstance)
{
    return(TRUE);                           // return success flag
}

//
// FrameWndProc --- callback function for application frame window.
// Searches frameMsgs[] for message match, runs corresponding function.
//
LONG FAR APIENTRY FrameWndProc(HWND hWnd, UINT wMsg, UINT wParam, LONG lParam)
{
    int i;                                  // scratch variable

    for(i = 0; i < dim(frameMsgs); i++)     // decode window message and
    {                                       // run corresponding function
        if(wMsg == frameMsgs[i].Code)
            return((*frameMsgs[i].Fxn)(hWnd, wMsg, wParam, lParam));
    }
```

Figure 2: **Here is the source code for the System Monitor utility.**

```
    return(DefWindowProc(hWnd, wMsg, wParam, lParam));
}

//
// DoCommand -- process WM_COMMAND message for frame window by
// decoding the menubar item with the menuitems[] array, then
// running the corresponding function to process the command.
//
LONG DoCommand(HWND hWnd, UINT wMsg, UINT wParam, LONG lParam)
{
    int i;                                  // scratch variable

    for(i = 0; i < dim(menuitems); i++)     // decode menu command and
    {                                       // run corresponding function
        if(wParam == menuitems[i].Code)
            return((*menuitems[i].Fxn)(hWnd, wMsg, wParam, lParam));
    }

    return(DefWindowProc(hWnd, wMsg, wParam, lParam));
}

//
// DoDestroy -- process WM_DESTROY message for frame window.
//
LONG DoDestroy(HWND hWnd, UINT wMsg, UINT wParam, LONG lParam)
{
    PostQuitMessage(0);                     // force WM_QUIT message to
    return(0);                              // terminate the event loop
}

//
// DoClose -- process WM_CLOSE message for frame window.
//
LONG DoClose(HWND hWnd, UINT wMsg, UINT wParam, LONG lParam)
{
    UpdateProfile();                        // save window size & position
    DestroyWindow(hWnd);                    // then close down app
    return(FALSE);
}

//
// DoVScroll -- process WM_VSCROLL message for frame window.
//
LONG DoVScroll(HWND hWnd, UINT wMsg, UINT wParam, LONG lParam)
{
    RECT rect;

    switch(LOWORD(wParam))                  // LOWORD vital for Win32
    {
        case SB_TOP:                        // go to top of output if
            if(CurLine)                     // we aren't there already
            {
                SetCurLine(0);
                Repaint();
            }
            break;

        case SB_BOTTOM:                     // go to bottom of output if
            if(CurLine < TopLine)           // we aren't there already
            {
                SetCurLine(TopLine);
                Repaint();
            }
            break;

        case SB_LINEUP:                     // scroll up by one line if
            if(CurLine)                     // we aren't already at top
            {
                SetCurLine(CurLine - 1);
                ScrollWindow(hWnd, 0, CharY, NULL, NULL);
                UpdateWindow(hWnd);
            }
            break;

        case SB_LINEDOWN:                   // scroll down by one line if
            if(CurLine < TopLine)           // we aren't already at bottom
            {
                SetCurLine(CurLine + 1);
                ScrollWindow(hWnd, 0, -CharY, NULL, NULL);
                GetClientRect(hWnd, &rect);
                rect.top = max(0, (LinesPerPage-1) * CharY);
                InvalidateRect(hWnd, &rect, TRUE);
                UpdateWindow(hWnd);
            }
            break;

        case SB_PAGEUP:                     // scroll up by one page
            SetCurLine(CurLine - LinesPerPage);
            Repaint();
            break;

        case SB_PAGEDOWN:                   // scroll down by one page
            SetCurLine(CurLine + LinesPerPage);
            Repaint();
            break;

        case SB_THUMBPOSITION:              // scroll display according
            SetCurLine(THUMBPOS);           // to new thumb position
            Repaint();
            break;
    }
    return(FALSE);
}

//
// DoPaint -- process WM_PAINT message for frame window.
//
LONG DoPaint(HWND hWnd, UINT wMsg, UINT wParam, LONG lParam)
{
    HDC hdc;
    PAINTSTRUCT ps;
    int i;

    hdc = BeginPaint(hWnd, &ps);            // get device context
    SelectObject(hdc, hFont);               // select non-prop. font

    for(i = 0; i < LinesPerPage; i++)       // paint lines of text
        PaintLine(hdc, i);                  // in the window

    EndPaint(hWnd, &ps);                    // release device context
    return(FALSE);
}

//
// DoSize -- process WM_SIZE message for frame window.
//
LONG DoSize(HWND hWnd, UINT wMsg, UINT wParam, LONG lParam)
{
    LinesPerPage = HIWORD(lParam) / CharY;  // window height / char height
    ConfigWindow();                         // calc display parameters
    if(CurLine > TopLine)                   // make sure window refilled
        SetCurLine(TopLine);                // if window got bigger
    return(0);
}

//
// DoMenuExit -- process File-Exit command from menu bar.
//
LONG DoMenuExit(HWND hWnd, UINT wMsg, UINT wParam, LONG lParam)
{
    SendMessage (hWnd, WM_CLOSE, 0, 0L);    // send window close message
    return(FALSE);                          // to shut down the app
}

//
// DoDisplayType -- process items on Display popup to select
// the type of information to display, then force window update
//
LONG DoDisplayType(HWND hWnd, UINT wMsg, UINT wParam, LONG lParam)
{
    HMENU hMenu;                            // scratch menu handle

    hMenu = GetMenu(hWnd);                  // update popup checkmark
    CheckMenuItem(hMenu, DisplayType, MF_UNCHECKED);
    DisplayType = wParam;
    CheckMenuItem(hMenu, DisplayType, MF_CHECKED);
    SendMessage(hWnd, WM_COMMAND, IDM_REFRESH, 0);  // update window
    return(FALSE);
}

//
// DoRefresh -- rebuild the information for display according to
// the currently selected display type, then refresh the window.
//
LONG DoRefresh(HWND hWnd, UINT wMsg, UINT wParam, LONG lParam)
{
    EmptyLines();                           // discard previous output

    switch(DisplayType)                     // call the appropriate
    {                                       // list walking routine
        case IDM_MODULE:                    // according to display type
            WalkModuleList();
            SetWindowCaption("Modules");
            break;

        case IDM_CLASS:
            WalkClassList();
            SetWindowCaption("Window Classes");
            break;

        case IDM_TASK:
            WalkTaskList();
            SetWindowCaption("Active Tasks");
            break;

        case IDM_HEAP:
            WalkGlobalHeap();
            SetWindowCaption("Global Heap");
            break;
    }

    ConfigWindow();                         // configure scroll bar etc.
    Repaint();                              // refresh the window

    return(FALSE);
```

```
}

//
// DoMenuAbout -- process File-About command from menu bar.
//
LONG DoMenuAbout(HWND hWnd, UINT wMsg, UINT wParam, LONG lParam)
{
    WNDPROC lpAboutProc;                    // scratch far pointer

    // allocate a thunk for the dialog callback, then display dialog
    lpAboutProc = MakeProcInstance((WNDPROC)AboutDlgProc, hInst);
    DialogBox(hInst, "AboutBox", hWnd, lpAboutProc);
    FreeProcInstance(lpAboutProc);
    return(FALSE);
}

//
// AboutDlgProc -- callback routine for About... dialog.  Basically
// ignores all messages except for the OK button, which dismisses dialog.
//
BOOL FAR APIENTRY AboutDlgProc (HWND hwnd, UINT msg, UINT wParam, LONG lParam)
{
    if((msg == WM_COMMAND) && (wParam == IDOK))
        EndDialog(hwnd, 0);                 // if OK button, destroy dialog
    else return(FALSE);                     // otherwise ignore message
}

//
// WalkModuleList -- uses ToolHelp functions to walk through
// module list and build formatted output in LinePtr[] array.
//
VOID WalkModuleList(VOID)
{
    MODULEENTRY me;                         // receives module info
    char temp[256];                         // scratch formatting buffer

    memset(&me, 0, sizeof(MODULEENTRY));    // initialize structure for
    me.dwSize = sizeof(MODULEENTRY);        // return of module data
    AddLine("Handle  Usage  Module    Pathname");   // format title
    ModuleFirst(&me);                       // initialize to 1st module

    do {                                    // format module information
        wsprintf(temp, "%04Xh   %4d   %-8.8s  %s", me.hModule, me.wcUsage,
            (LPSTR) me.szModule, (LPSTR) me.szExePath);
        AddLine(temp);                      // add to array for output
    } while(ModuleNext(&me));               // get next module name
}

//
// WalkClassList -- uses ToolHelp functions to walk through
// window class list and build formatted output in LinePtr[] array.
//
VOID WalkClassList(VOID)
{
    CLASSENTRY ce;                          // receives window class info
    char temp[256];                         // scratch formatting buffer

    memset(&ce, 0, sizeof(CLASSENTRY));     // initialize structure for
    ce.dwSize = sizeof(CLASSENTRY);         // return of window class data
    AddLine("Owner   Class Name");          // format title
    ClassFirst(&ce);                        // initialize to 1st class

    do {                                    // format owner & classname
        wsprintf(temp, "%04Xh   %s", ce.hInst, (LPSTR) ce.szClassName);
        AddLine(temp);                      // add to array for output
    } while(ClassNext(&ce));                // get next class name
}

//
// WalkTaskList -- uses ToolHelp functions to walk through
// task list and build formatted output in LinePtr[] array.
//
VOID WalkTaskList(VOID)
{
    TASKENTRY te;                           // receives task info
    MODULEENTRY me;                         // receives module info
    char temp[256];                         // scratch formatting buffer

    memset(&te, 0, sizeof(TASKENTRY));      // initialize structure for
    te.dwSize = sizeof(TASKENTRY);          // return of task info
    memset(&me, 0, sizeof(MODULEENTRY));    // initialize structure for
    me.dwSize = sizeof(MODULEENTRY);        // return of module data
    AddLine("Task     Parent   Instance   Module   Module     Module");
    AddLine("Handle   Task     Handle     Handle   Name       Pathname");
    TaskFirst(&te);                         // initialize to 1st task

    do {                                    // format task information
        ModuleFindHandle(&me, te.hModule);  // get modulename & pathname
        wsprintf(temp, "%04Xh    %04Xh    %04Xh      %04Xh    %-8.8s   %s",
            te.hTask, te.hTaskParent, te.hInst, te.hModule,
            (LPSTR) me.szModule, (LPSTR) me.szExePath);
        AddLine(temp);                      // add to array for output
    } while(TaskNext(&te));                 // get next task
}

//
// WalkGlobalHeap -- uses ToolHelp functions to walk through
// global heap and build formatted output in LinePtr[] array.
//
VOID WalkGlobalHeap(VOID)
{
    GLOBALENTRY ge;                         // receives heap block info
    TASKENTRY te;                           // receives task info
    MODULEENTRY me;                         // receives module info
    char temp[256];                         // scratch formatting buffer

    memset(&ge, 0, sizeof(GLOBALENTRY));    // initialize structure for
    ge.dwSize = sizeof(GLOBALENTRY);        // return of heap block info
    memset(&me, 0, sizeof(MODULEENTRY));    // initialize structure for
    me.dwSize = sizeof(MODULEENTRY);        // return of module data
    memset(&te, 0, sizeof(TASKENTRY));      // initialize structure for
    te.dwSize = sizeof(TASKENTRY);          // return of task info
    AddLine("Handle   Linear Addr    Size     Type   Owner"); // format title
    GlobalFirst(&ge, GLOBAL_ALL);           // initialize to 1st block

    do {                                    // format heap block info
        if(TaskFindHandle(&te, ge.hOwner))  // get owner's name
            ModuleFindHandle(&me, te.hModule);
        else if(!ModuleFindHandle(&me, ge.hOwner))
            me.szModule[0] = '\0';
        wsprintf(temp, "%04Xh     %08lXh   %08lXh  %04Xh  %04Xh %s",
            ge.hBlock, ge.dwAddress, ge.dwBlockSize, ge.wType,
            ge.hOwner, (LPSTR) me.szModule);
        AddLine(temp);                      // add to array for output
    } while(GlobalNext(&ge, GLOBAL_ALL));   // get next class name
}

//
// SetCurLine - called to set CurLine to valid value, clamped to
// the range (0...TopLine), and redraw thumb on scroll bar.
//
VOID SetCurLine(int NewLine)
{
    CurLine = min(max(NewLine, 0), TopLine);
    SetScrollPos(hFrame, SB_VERT, CurLine, TRUE);
}

//
// ConfigWindow -- Configures various display parameters and scrollbar
// according to total lines of output, current window size, and the
// number of lines that will fit into the window.
//
VOID ConfigWindow(VOID)
{
    // calc line number of first line of last page
    TopLine = max(TotLines - LinesPerPage,0);

    // update scroll bar range and thumb position
    SetScrollRange(hFrame, SB_VERT, 0, TopLine, FALSE);
    SetScrollPos(hFrame, SB_VERT, CurLine, TRUE);
}

//
// AddLine -- called with a pointer to an ASCIIZ string, allocates
// memory from the heap to hold the string, puts the pointer
// to the heap block into the next position in the LinePtr[] array,
// and updates the total line count.
//
VOID AddLine(char * p)
{
    char * q;                               // scratch pointer

    if(TotLines == MAXLINES)                // bail out if line pointer
        return;                             // array is already full
    q = malloc(strlen(p)+1);                // allocate memory for line
    if(q == 0)                              // bail out out if no
        return;                             // heap space available
    strcpy(q, p);                           // copy string to heap
    LinePtr[TotLines] = q;                  // put heap pointer into array
    TotLines++;                             // count lines of output
}

//
// EmptyLines - releases all heap blocks in LinePtr[] array,
// then zeros out the line pointers and the total line count
//
VOID EmptyLines(VOID)
{
    int i;                                  // scratch variable

    for(i = 0; i < MAXLINES; i++)
    {
        if(LinePtr[i])                      // if this position in
        {                                   // the LinePtr array is
            free(LinePtr[i]);               // nonzero, release the
            LinePtr[i] = NULL;              // heap block, then zero
        }                                   // out the LinePtr slot
    }

    CurLine = 0;                            // initialize various
    TotLines = 0;                           // other global variables
    TopLine = 0;
}

//
// PaintLine -- paint a single line of text in the window.
```

SYSMON.C
4 of 4

```
// The passed line number is relative to the window, NOT to the
// total array of formatted output available to be painted.
//
VOID PaintLine(HDC hdc, INT RelLine)
{
    int Line = RelLine + CurLine;
    if(LinePtr[Line])
        TextOut(hdc, 0, RelLine*CharY, LinePtr[Line], strlen(LinePtr[Line]));
}

//
// Repaint - force repaint of all formatted output in main window
//
VOID Repaint(VOID)
{
    InvalidateRect(hFrame, NULL, TRUE);     // force repaint entire window
}

//
// SetWindowCaption -- concatenate the application name with the
// display type, then update the frame window's title bar.
//
VOID SetWindowCaption(char * szDisplayType)
{
    char szTemp[256];                       // scratch buffer

    strcpy(szTemp, szAppName);              // get application name
    strcat(szTemp, " - ");                  // add separator
    strcat(szTemp, szDisplayType);          // add information type
    SetWindowText(hFrame, szTemp);          // put result into title bar
}

//
// UpdateProfile() --  saves the current window size and position
// and display type in the application's private INI file.
//
VOID UpdateProfile(VOID)
{
    RECT rect;
    char temp[20];

    if(IsIconic(hFrame) || IsZoomed(hFrame)) return;

    GetWindowRect(hFrame, &rect);

    wsprintf(temp,"%d", rect.left);
    WritePrivateProfileString("Frame", "xul", temp, szIni);

    wsprintf(temp,"%d", rect.top);
    WritePrivateProfileString("Frame", "yul", temp, szIni);

    wsprintf(temp,"%d", rect.right);
    WritePrivateProfileString("Frame", "xlr", temp, szIni);

    wsprintf(temp,"%d", rect.bottom);
    WritePrivateProfileString("Frame", "ylr", temp, szIni);

    wsprintf(temp,"%d", DisplayType);
    WritePrivateProfileString("Frame", "type", temp, szIni);
}

//
// TimerProc() -- Callback for 10 second timer. Refresh display
// if window is not minimized and does not have the focus.
//
WORD FAR APIENTRY TimerProc(HWND hwnd, WORD message, WORD wParam, LONG lParam)
{
    if((!IsIconic(hFrame)) && (hFrame != GetFocus()))
        SendMessage(hFrame, WM_COMMAND, IDM_REFRESH, 0);
    return(FALSE);
}
```

SYSMON's frame window—and the message is handled exactly as though the user had clicked the Refresh! item on the menu bar. This is a potent example of how exploitation of Windows' event-driven, message-based architecture can simplify application design.

The third (and perhaps most novel) characteristic of SYSMON's source code I'd like to bring to your attention is the handling of the text displayed in the main window. If you try to integrate the painting of text and the control of the scrollbar's range and thumb position with the code that actually walks the system lists, you get a real mess. There's no way to predict how many items will be found in a list in advance, and this may change from one walk of the list to another, making scrolling the display backwards and forwards and maintenance of the scrollbar itself a messy proposition. The solution is to completely divorce the code that generates the text to be displayed from the code that actually paints the text in the window. The two bodies of code communicate via an array of pointers called LinePtrs[]. Each slot in LinePtrs[] corresponds to a line of text to be displayed and either contains a pointer to an ASCIIZ string or a NULL pointer if the line is empty.

The central routines that format information for display—that is, the four routines WalkClassList(), WalkTaskList(), WalkModuleList(), and WalkGlobalHeap()—initialize the display (from their point of view) by calling EmptyLines(), and place text on the display by repeatedly calling AddLine(). But what EmptyLines() actually does is just zero out LinePtrs[] and the count of the lines to be displayed, while AddLine() merely allocates some heap space to hold the string that is passed to it, puts a pointer to the heap block into LinePtrs[], and increments the line count. After a list-walking routine is done with its work, it calls ConfigWindow(), which calculates the line number of the first line in the last display page (if there is enough text tha scrolling will be required) and configures or hides the scrollbar.

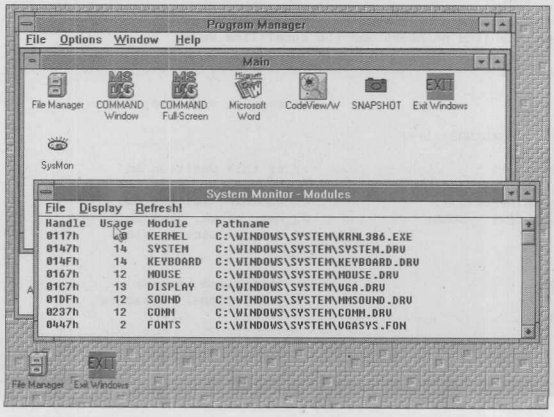


Figure 3: **The SYSMON utility, with the task list display selected.**

Which routine takes text out of LinePtrs[] and paints it in the SYSMON window? Why, DoPaint(), naturally, which is called whenever the application receives a WM_PAINT message. DoPaint() can behave as though the contents of LinePtrs[] are static, and it gets the total number of lines of text from the variable TotLines, the number of the first line to be displayed in the window from the variable CurLine, and the number of lines that the window can hold from the variable LinesPerPage. The message-handler routine for scrollbar messages, DoVScroll(), needs to do little more than manipulate the variable CurLine and send a WM_PAINT message to the SYSMON window. At that point, the display logic degenerates to mindless bookkeeping, which is certainly a curious event in Windows programming.

THE IN-BOX Please send your questions, comments, and suggestions to me at any of these electronic mail addresses:
PC MagNet: 72241,52
MCI Mail: rduncan
BIX: rduncan □

POWER PROGRAMMING

The Windows 3.1 ToolHelp DLL, Part 3

BY RAY DUNCAN

For the last two columns, we've been looking at Windows 3.1's TOOLHELP.DLL, a dynamic link library that gives you access to previously undocumented Windows data structures and function calls. Two sample programs described in the August 1992 issue—BADAPP and BADAPP2—showed how to implement a notification callback routine that will be called by TOOLHELP if your program makes a function call with an invalid parameter. SYSMON, published in the September 15, 1992, issue, demonstrated how to use the TOOLHELP functions to walk the system module list, task list, window class list, and global heap.

In this installment, we'll examine the TOOLHELP functions that let your program intercept certain interrupts—such as divide by zero, general protection (GP) fault, breakpoint, and invalid opcode—in a "well-behaved" manner. The ability to capture and service these interrupts is vital for anyone who wants to write a Windows-based debugger, but it's also handy if you're writing any sort of high-level language interpreter or even an application with a built-in macro language. Any time you place a programming facility—especially one that supports pointers—in the hands of an end-user, you can no longer hope to test your application exhaustively, so you must fight back with robust error handling.

INTERRUPTS AND THEIR KIN As every English schoolboy knows, the Intel 80x86 family of processors supports 256 levels of interrupts. The location of the interrupt service routine (ISR), or interrupt handler, for each interrupt is specified in a RAM-based table that is created and updated by software and interpreted by hardware. In real mode, the table is called the interrupt vector table; it is always located at address 0000:0000 and consists of 256 far pointers, or 1,024 bytes. In protected mode, on the other hand, the table is called the Interrupt Descriptor Table (IDT). An IDT is more complex than a real-mode vector table, it can be located anywhere in memory, and it can be maintained on a per-task basis. When an interrupt occurs, the CPU is hardwired to push the current contents of the CPU flags and instruction pointer (CS:IP) onto the stack, fetch the address of the interrupt handler from the appropriate table according to the current execution mode, and load that address into the instruction pointer. When the handler routine is finished, it executes an interrupt return (IRET) instruction, and execution resumes at the point of interrupt.

Our examination of ToolHelp continues with a look at the functions that let applications intercept interrupts in a well-behaved manner.

So far, so good. But this explanation doesn't take into account that there are two basic classes of interrupts. *External* interrupts are the kind we generally think of when we hear the word interrupt; they are generated by adapter cards, clock chips, and other gizmos located outside the CPU chip. These devices usually interrupt the CPU by signaling it through the INTR pin; if the CPU accepts the interrupt by a pulse on the "interrupt acknowledge" (INTA) pin, the external device tells the CPU which interrupt handler to use by jamming a number in the range 0 to 255 onto the CPU's data bus. (There is also a nonmaskable interrupt pin, NMI, which we'll ignore here.)

In real mode, a program can mask—or block the reception of—external interrupts by executing the DI (disable interrupts) instruction, and can unmask interrupts by executing the EI (enable interrupts) instruction. In protected mode, execution of EI and DI is mostly reserved for the operating system and its device drivers, though certain protected mode applications with I/O privilege level (IOPL) may also be allowed to execute the instructions, and real mode applications running in a Virtual 86 machine will get special treatment.

Internal interrupts, on the other hand, are triggered by events inside the CPU—the execution of an instruction, or an error condition that arises during the execution of an instruction. The one we're all familiar with is the INT instruction, which can be used in real mode to force the occurrence of any of the 256 possible interrupts. MS-DOS uses INT as its application program interface mechanism: When DOS boots up, it puts the address of its entry point into slot 21h of the interrupt vector table, and programs request services from DOS by loading various parameters into registers and then executing an INT 21h instruction to transfer control to DOS's function dispatcher.

The neatest thing about the INT instruction, aside from its use as a sort of compact FAR CALL instruction, is that it makes the initial testing of handlers for external interrupts much easier: You can

it with a small test program that executes the appropriate INT instruction under a variety of conditions, even if the external hardware that will ultimately generate the interrupt has not yet been built.

But there's a lot more to internal interrupts than the INT instruction. There's the special 1-byte debugger breakpoint opcode, 0CCh, which always generates an interrupt 3. There's interrupt 1, the single-step interrupt also designed especially for use by debuggers. And finally, there's the whole host of interrupts (known by the specific technical term *exceptions*) listed in Figure 1 that occur when a program tries to execute an instruction with the wrong parameters or an instruction it isn't entitled to use.

Most of these exceptions are of interest only to the authors of protected mode operating systems, and their implementation varies from one model of the Intel 80x86 series to another. When they occur in the context of an application program, the operating system's response is usually to terminate the application with extreme prejudice and put a discouraging message up on the screen to let the user know he just lost all his work. For a more detailed explanation, consult *Programming the 80386* by Crawford and Gelsinger (Sybex Inc., 1987), or *Programmer's Reference Manual: The Processor and Coprocessor* by Robert Hummel (Ziff-Davis Press, 1992). Note that exceptions are further subdivided into two groups: faults and

Exceptions

Interrupt	Description
0	Divide by zero
4	Overflow
5	BOUND range exceeded
6	Invalid opcode
7	Coprocessor not available
8	Double-fault
9	Coprocessor segment overrun
10 (0Ah)	Invalid task segment
11 (0Bh)	Segment not present
12 (0Ch)	Stack fault
13 (0Dh)	General protection fault
14 (0Eh)	Page fault
16 (10h)	Coprocessor error
17 (11h)	Alignment check

Figure 1: **These interrupts, known as exceptions, occur when a program tries to execute invalid instructions.**

essentially that faulting instructions are restartable; trapping instructions are not.

When an instruction faults, the error condition is detected by the CPU before any part of the instruction executes, and the address of the "bad" instruction itself is pushed onto the stack. This gives the interrupt handler the opportunity to recover from the error condition: It can retrieve the address of the faulting instruction from the stack, decode the instruction type, examine the registers or memory being accessed by the instruction, make any changes that are necessary, and, finally, restart execution of the failed instruction by executing an IRET. At the least, the handler for a fault exception can gather and display very detailed diagnostic information. In contrast, a trap exception isn't detected until after the error has occurred, and the CPU pushes the address of the instruction after the failed instruction onto the stack. Because Intel 80x86 instructions have varying lengths, and the trapped instruction may have already changed registers and/or memory, recovering from a trap exception would be a nasty proposition. Fortunately, most Intel 80x86 exceptions are faults.

HANDLING FAULTS TOOLHELP.DLL provides applications with access to a subset of the possible exceptions: interrupts 0, 1, 3, 6, 12, 13, and 14. For most applications, only two are worrisome: Interrupt 0 (divide by zero) and Interrupt 13 (general protection fault). Interrupt 0 is triggered if your program executes either a division by zero operation or a division that results in an overflow; that is, the quotient will not fit into the destination register. Interrupt 13 is triggered by a wide variety of conditions, the most common of which are use of an invalid selector (such as dereferencing an uninitialized or NULL far pointer) or using a valid selector but an offset that lies outside the designated segment's "limit" (which is from 1 to 65536 bytes).

The framework for application exception handling with TOOLHELP is quite similar to the technique we used for the handling of parameter validation errors in BADAPP2.C:

- The program must contain a callback routine, which is entered from TOOLHELP when an exception is detected.
- The callback routine must be
- The initialization code for the applic. tion must allocate a thunk for the callback routine with MakeProcInstance(), and register the callback with TOOLHELP by calling InterruptRegister().
- The termination code for the application must notify TOOLHELP that it no longer wants to handle exceptions by calling InterruptUnRegister(), and then free the callback thunk by calling FreeProcInstance().

When the application's callback routine is entered, information about the cause of the exception is available on the stack (Figure 2). The state of all general registers at entry is indeterminate, except that AX has been loaded by the thunk code with the selector for the application's DGROUP. After examining the stack, the callback routine can take one of four possible actions:

- Indicate that it doesn't want to handle the exception by executing a RETF instruction; the exception will then be processed by the next handler on the chain (usually TOOLHELP itself).
- Fix the cause of the fault, discard th first 10 bytes of the stack, and execute an IRET instruction to restart execution of the faulted instruction.
- Call TerminateApp() to kill the faulting application.
- Retain control by initializing all general and segment registers to valid values and then branching to an error-handling routine in the application, never executing a RETF or IRET.

Due to the design of the TOOLHELP interface, some of this is tough to implement in a high-level language, so Win-

The Stack After an Exception

. (application's stack)

Flags (fault)	SP+0EH
CS (fault)	SP+0CH
IP (fault)	SP+0AH
TOOLHELP handle (internal)	SP+08H
Interrupt Number	SP+06H
AX (at point of fault)	SP+04H
CS (TOOLHELP)	SP+02H
IP (TOOLHELP)	SP+00H

Figure 2: **This shows the stack contents at entry to an application interrupt callback routine.**

dows application interrupt callbacks are ypically written either entirely in assembly language or as an assembly language stub that in turn calls a C routine to do the body of the work.

And now we must turn to the truly horrid aspect of the TOOLHELP interface. Because Windows does not take full advantage of the Intel 80x86 protection architecture and runs all applications on the same Local Descriptor Table (LDT), Windows has no way to dispatch exceptions on a per-application basis. As a result, when your application registers an interrupt callback routine, that routine will receive callbacks for all exceptions that occur in the system—whether or not your own application caused them. This means that your application must maintain internal flags that can be tested by the callback to determine whether the exception should be handled or ignored, and update these flags each time the application gains or yields control of the

BADAPP3.C
Partial Listing

```
// BADAPP3 - Application Error Testbed for Windows 3.1 with Notification
//          Callback and GP Fault/Divide by Zero Interrupt Handlers
// Copyright (C) 1992 Ray Duncan
// Ziff Davis Publishing * PC Magazine

#define dim(x) (sizeof(x) / sizeof(x[0]))   // returns no. of elements

#include "windows.h"
#include "toolhelp.h"
#include "badapp.h"

HANDLE hInst;                               // module instance handle
HWND hFrame;                                // handle for frame window
WNDPROC lpNotifyCallback;                   // NotifyCallback thunk addr
WNDPROC lpIntCallback;                      // IntCallback thunk addr

UINT ActiveFlag = FALSE;                    // TRUE when handler active
CATCHBUF CatchBuf;                          // holds Catch() state
LPCATCHBUF lpCatchBuf = &CatchBuf;          // far pointer for Throw()
UINT IntNum;                                // interrupt number
LPVOID lpFault;                             // address of fault

char szShortAppName[] = "BadApp";           // short application name
char szAppName[] = "BadApp - UAE Tester";   // long application name
char szMenuName[] = "BadAppMenu";           // name of menu resource
char szIconName[] = "BadAppIcon";           // name of icon resource

struct decodeWord {                         // structure associates
    UINT Code;                              // messages or menu IDs
    LONG (*Fxn)(HWND, UINT, UINT, LONG); }; // with a function

//
// Table of window messages supported by FrameWndProc()
// and the functions which correspond to each message.
//
struct decodeWord messages[] = {
    0, DoNotifyMessage,
    0, DoIntMessage,
    WM_PAINT, DoPaint,
    WM_COMMAND, DoCommand,
    WM_DESTROY, DoDestroy, } ;

//
// Table of menubar item IDs and their corresponding functions.
//
struct decodeWord menuitems[] = {
    IDM_EXIT, DoMenuExit,
    IDM_GPFAULT, DoMenuGPFault,
    IDM_HANG, DoMenuHang,
    IDM_ZERODIV, DoMenuZeroDiv,
    IDM_BADHDC, DoMenuBadHDC,
    IDM_BADPTR, DoMenuBadPtr, } ;

  .
  .  // WinMain() omitted here - same as BADAPP2
  .

//
// InitInstance --- instance initialization code for this application.
//
BOOL InitInstance(HANDLE hInstance, int nCmdShow)
{
    hFrame = CreateWindow(                  // create frame window
        szShortAppName,                     // window class name
        szAppName,                          // text for title bar
        WS_OVERLAPPEDWINDOW,                // window style
        CW_USEDEFAULT, CW_USEDEFAULT,       // default position
        CW_USEDEFAULT, CW_USEDEFAULT,       // default size
        NULL,                               // no parent window
        NULL,                               // use class default menu
        hInstance,                          // window owner
        NULL);                              // unused pointer

    if(!hFrame) return(FALSE);              // error, can't create window

    ShowWindow(hFrame, nCmdShow);           // make frame window visible
    UpdateWindow(hFrame);                   // force WM_PAINT message

    // allocate private message numbers for use by callbacks
    messages[0].Code = RegisterWindowMessage("BADAPP_NOTIFY");
    messages[1].Code = RegisterWindowMessage("BADAPP_INTERRUPT");

    // allocate thunks for callbacks
    lpNotifyCallback = MakeProcInstance((WNDPROC) NotifyCallback, hInst);
    lpIntCallback = MakeProcInstance((WNDPROC) ISR, hInst);

    // register parameter validation and interrupt callbacks
    NotifyRegister(NULL, lpNotifyCallback, NF_NORMAL);
    InterruptRegister(NULL, lpIntCallback);

    return(TRUE);                           // return success flag
}

//
// TermInstance -- instance termination code for this application.
//
BOOL TermInstance(HANDLE hinstance)
{
    NotifyUnRegister(NULL);                 // free validation callback
    InterruptUnRegister(NULL);              // free interrupt callback
    FreeProcInstance(lpNotifyCallback);     // release callback thunks
    FreeProcInstance(lpIntCallback);
    return(TRUE);                           // return success flag
}

//
// FrameWndProc --- callback function for application frame window.
//
LONG FAR APIENTRY FrameWndProc(HWND hWnd, UINT wMsg, UINT wParam, LONG lParam)
{
    int i;                                  // scratch variables
    long l;

    ActiveFlag = TRUE;                      // turn on exception handling

    if(Catch(lpCatchBuf))                   // save context, or...
    {
        // if regaining control from Throw(), exception occurred during
        // message handling.  Send exception message, turn off exception
        // handling, and pass message to default handler.
        PostMessage(hFrame, messages[1].Code, 0, 0);
        ActiveFlag = FALSE;
        return(DefWindowProc(hWnd, wMsg, wParam, lParam));
    }

    for(i = 0; i < dim(messages); i++)      // decode window message and
    {                                       // run corresponding function
        if(wMsg == messages[i].Code)
        {
            l = (*messages[i].Fxn)(hWnd, wMsg, wParam, lParam);
            ActiveFlag = FALSE;             // turn off exception handling
            return(l);                      // return message result
        }
    }

    // if no match in messages[] table, turn off exception handling
    // and pass message to default message handler
    ActiveFlag = FALSE;
    return(DefWindowProc(hWnd, wMsg, wParam, lParam));
}

  .
  .  // DoCommand(), DoPaint(), and various routines
  .  // omitted here - see BADAPP and BADAPP2
  .

//
// DoIntMessage -- process private message from interrupt handler
//
LONG DoIntMessage(HWND hWnd, UINT wMsg, UINT wParam, LONG lParam)
{
    char temp[256];                         // formatting buffer

    // format and display interrupt number and fault address
    wsprintf(temp, "Interrupt %02Xh at %04X:%04Xh", IntNum,
        HIWORD(lpFault), LOWORD(lpFault));
    MessageBox(hWnd, temp, "BadApp", MB_OK | MB_ICONEXCLAMATION);

    return(FALSE);
}

//
// InterruptCallback -- callback routine for faults & exceptions.
// Entered from "umbrella interrupt handler" in ISR.ASM.  Restores
// execution context to that saved in the latest call to Catch().
//
VOID FAR APIENTRY InterruptCallback(UINT Num, LPVOID Addr)
{
    IntNum = Num;                           // save interrupt number
    lpFault = Addr;                         // and fault address
    Throw(lpCatchBuf, 1);                   // then restore old context
}
```

Figure 3: **Extracts from BADAPP3.C showing the portions concerned with exception handling. The full source code can be downloaded from PC MagNet.**

CPU. This becomes a major maintenance problem in a hurry, particularly for complex applications that are implemented as a suite of discrete tasks communicating via DDE or SendMessage().

AN EXAMPLE To illustrate the fundamental steps involved with TOOLHELP-based application exception handling under Windows, I've modified BADAPP2 to create a new program—BADAPP3—that includes an interrupt callback routine. From a user's point of view, BADAPP3 looks just like BADAPP and BADAPP2, except that when the user picks GP Fault, Divide by Zero, or Pass Bad Pointer on the Crash pop-up menu, he sees a diagnostic dialog box that originates in BADAPP3, rather than one displayed by Dr. Watson or the Windows kernel. And BADAPP3 cannot be terminated unilaterally by the operating system. Extracts from the source code for BADAPP3 that are relevant to the handling of exceptions are shown in Figure 3, and the assembly language source code for the callback stub is shown in Figure 4. A screen shot of BADAPP3 in action is shown in Figure 5. You can download the complete source code for BADAPP3, along with the precompiled executable program (archived as BADAP3.ZIP) from PC MagNet.

The overall structure of BADAPP3 is table-driven and straightforward, but there are two areas to note. The first is the method by which we keep control and restore the CPU to a known state when our callback routine is called because of a divide by zero or general protection fault that occurred in our application. When the callback is entered from TOOLHELP, we can figure out where the fault occurred easily enough, but the register contents and especially the depth of the stack are unpredictable. Executing a Goto to some error routine in our program would not be an adequate response, mainly because it wouldn't reset the stack and several faults in a row might cause the stack to overflow (resulting in yet another, more difficult to handle exception). However, we can easily overcome this problem by exploiting the little-known Windows functions Catch() and Throw().

Catch() and Throw() form a general-purpose mechanism for structured exception handling. You call Catch() at some convenient central point in your program to save the current execution environment, including all CPU registers in a structure of type CATCHBUF. When Catch() is invoked in this way, it returns a value of zero. At some future point in the execution of your program, when you find yourself in a hopeless situation, you can call Throw() with the address of the CATCHBUF structure and an arbitrary value that can be anything but zero. Then something magical happens: The CPU state including the stack pointer is restored according to the contents of the CATCHBUF structure, and execution continues by "returning" from the original call to Catch()—except that the value of Catch() this time is whatever you passed to Throw(). You distinguish the two possible types of return from Catch() by framing Catch() in an if{} structure:

```
CATCHBUF CatchBuf;
  .
  .
  .

if(Catch(&CatchBuf)
{
  // we got here
  // by a Throw()
}
```

ISR.ASM

Complete Listing

```
; ISR.ASM - umbrella interrupt handler for BADAPP3
; Copyright (C) 1992 Ray Duncan
; PC Magazine * Ziff Davis Publishing

        extrn   INTERRUPTCALLBACK:far

DGROUP  group   _DATA

_DATA   segment word public 'DATA'

        extrn   _ActiveFlag:word

_DATA   ends

_TEXT   segment word public 'CODE'

        assume  cs:_TEXT,ds:DGROUP

        public  ISR
ISR     proc    far

        push    ds
        push    bp
        mov     bp,sp                   ; point to stack frame
        mov     ds,ax                   ; make DGROUP addressable

        test    _ActiveFlag,-1          ; is interrupt handling active?
        jz      ISR2                    ; no, jump

        cmp     word ptr [bp+0ah],0     ; check for Int 0
        je      ISR1                    ; (divide by zero) or
        cmp     word ptr [bp+0ah],0dh   ; Int 0DH (GP fault)
        jne     ISR2                    ; jump if neither of these

ISR1:   push    [bp+0ah]                ; push interrupt number and
        push    [bp+10h]                ; far pointer to faulting
        push    [bp+0eh]                ; instruction

        call    INTERRUPTCALLBACK       ; transfer to C routine

ISR2:   pop     bp                      ; if not GP fault or divide by
        pop     ds                      ; zero, let TOOLHELP handle it
        ret

ISR     endp

_TEXT   ends

        end
```

Figure 4: **The assembly language stub entered by a far call from TOOLHELP when an exception is detected. This code checks whether the exception should be serviced, and if so, passes control to the main handler written in C.**

Figure 5: **This is the dialog box displayed by BADAPP3 when a general protection fault is detected.**

```
else
{
  // this was a return
  // from an intentional
  // call to Catch()
}
  .
  .
  .
// bad error detected,
// bail out of our current
// hopeless state
Throw(&CatchBuf, 1);
```

The call to Catch() in BADAPP3 is located right in the frame window's message handler. Since Windows programs are message-driven, we can safely assume that a divide by zero or general protection fault is going to occur in some code that executes in response to a message. Therefore, we call Catch() immediately before decoding a message and handing it off to any internal subroutine. If a fault occurs, our assembly language stub, ISR, is entered, which in turn calls the C routine InterruptHandler(). InterruptHandler() saves the interrupt number and location of the faulting instruction for later display, then calls Throw(), and the CPU reverts to its state at Catch(). We are then free to back out of the situation by posting a private message to ourselves indicating that an error occurred, then handing the message to Windows' default message handler (DefWindowProc) instead of our own handler (since our own handler is now known to fault on the same message). By the time we have processed the private error message, the application state has been cleaned up enough so that we can safely display a dialog box about the error.

The other aspect of BADAPP3 you may want to ponder and critique is the strategy for letting the exception callback know whether the application is currently in control. BADAPP3 has a global, static-variable, called ActiveFlag, which is updated by the body of the program and checked by the stub routine ISR. I initially considered turning ActiveFlag on and off in the program's main event loop; that is, turning ActiveFlag off before a call to GetMessage(), and turning it on again upon return from GetMessage(). Then it occurred to me that in the worst case (though not in this particular program), forced execution of another task that might be responsible for an exception could quite easily be hidden within the call to DispatchMessage()—for example, if one of the application's window message handlers called SendMessage(). Therefore, I decided to maintain the state of ActiveFlag within the window message callback routine instead, setting the flag to TRUE whenever an internal routine is explicitly run in response to a message, and setting it to FALSE before calling DefWindowProc or returning from the message callback. This approach could easily be extended by adding a new member to the messages[] structure that defines the state of ActiveFlag—and thus the handling of exceptions—on a per-message basis.

FURTHER READING A very helpful tutorial on TOOLHELP.DLL exception handling, written by Kraig Brockschmidt of Microsoft Developer Relations, can be downloaded from the WINSDK forum on CompuServe in the file FAULTW.ZIP. I found the technique of using Catch() and Throw() in a Windows application exception handler in this document.

THE IN-BOX Please send your questions, comments, and suggestions to me at any of the following electronic mail addresses:

PC MagNet: 72241,52
MCI Mail: rduncan
BIX: rduncan
Internet: duncan@csmcmvax.bitnet □

Computers, FCC Class A, Class B, and You — or When is it better to get a B than an A?

You need to know the difference between computers that meet the FCC class B radio frequency emissions standards and those that meet only the Class A standards.

Computers emit radio signals in their operation. Because these signals may cause interference to radio and television reception, the marketing and the use of computers is regulated by the Federal Communications Commission. Under federal rules, computer users are responsible for remedying interference, including interference in neighboring homes.

Computers certified by the FCC as meeting the Class B standard are less likely to cause interference to radio and TV reception than those that have been verified by the manufacturer or importer to the Class A standards. Only Class B certified computers may be advertised, sold, or leased for use in residences. A similar regulatory program applies in Canada.

Buyers seeking computers for use in homes (including offices at home) should shop for computers and peripherals which have been Class B certified. These devices carry a label with an FCC ID number. Both new and used Class A verified devices may be sold only for use in commercial and industrial locations. Signals from computers are more likely to be masked by electrical noise from other equipment in such an environment. These areas are also likely to have fewer radios and TVs. Accordingly, equipment marketed only for use in these locations may meet the less rigorous Class A standard. Class B certified equipment may be marketed for use in residences as well as commercial and industrial locations.

As you shop for a computer for use in your home, look for the FCC classification in the specifications or ask your vendor to recommend only machines that have been certified to the Class B limits. TV viewers and radio listeners in your home and in neighboring homes will be glad you did.

Simply Faster.

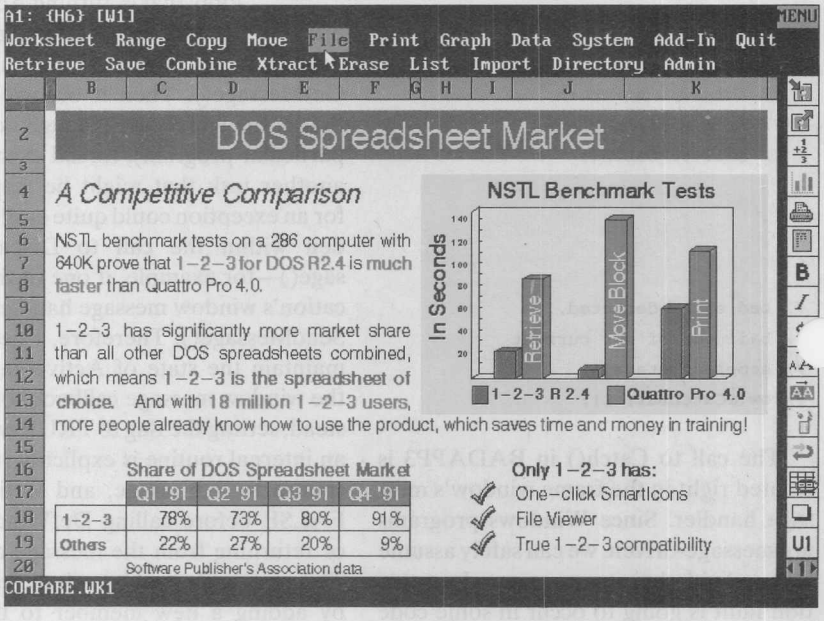
1-2-3 for DOS Release 2.4.

1-2-3® is not only easier to learn and use, it's simply faster than any other DOS spreadsheet, including Quattro® Pro Release 4.0, especially on IBM® XTs and 286 machines.

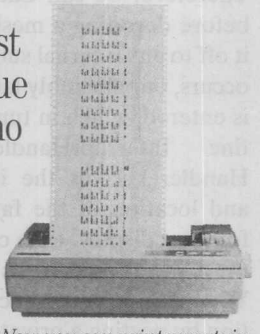
It's the only DOS spreadsheet to offer SmartIcons™—a unique palette of over 75 icons, 12 of which are customizable, giving you instant, one-click access to your most commonly performed spreadsheet tasks. SmartIcons give you all the ease-of-use and graphical advantages of our Windows™ applications, without the need to change operating systems. And 1-2-3 for DOS also gives you such unique 1-2-3 features as file Viewer, Auditor and Backsolver, a quicker and more efficient goal-seeker. And only 1-2-3 for DOS offers true compatibility with all versions of 1-2-3 across platforms.

Only 1-2-3 offers SmartIcons for true one-click access to virtually all your spreadsheet tasks, unlike the limited SpeedBar™ in Quattro Pro which requires you to work through a menu tree.

And right now, you can get both 1-2-3 for DOS Release 2.4 and Freelance Graphics® for DOS Release 4.0, our presentation graphics package, together for just $229—a suggested retail value of $645. For a free auto demo or to order your upgrade directly from Lotus,® call* **1-800-TRADEUP, Ext. 7014.** Or visit your Lotus Authorized Reseller.



*Performance tests prove that 1-2-3 for DOS Release 2.4 beats Quattro Pro 4.0 in areas you care most about.***



Now you can print reports in landscape mode on all printers, including dot matrix printers.

Lotus 1-2-3 for DOS

*In Canada, call **1-800-GO-LOTUS.** Please have your credit card and product package ready when you call. **NSTL standards used in tests performed on an IBM AT® 286 with 640K memory.